F2C-ACC Users Guide

Compiler Version 4.2

Developed by: Mark Govett
National Oceanic and Atmospheric Administration (NOAA)
Earth System Research Laboratory (ESRL)
August 2012

Additional information and user support:
http://www.esrl.noaa.gov/gsd/ab/ac/Accelerators.html
help.accel.gsd@noaa.gov

F2C –ACC is a translation tool used to convert Fortran code into C or CUDA. It was designed to support a programming model in which Fortran code, running on the CPU, makes calls to routines that are run on the GPU. These routines are translated by F2C-ACC into CUDA, the language used by NVIDIA GPUs. The C code generation is useful for testing and debugging, and as a basis for OpenCL generation. F2C-ACC generated code is processed by the m4 processor to resolve inter-language differences between Fortran and C / CUDA on the targeted machine. The translator supports both running a single routine on the GPU (**accelerator style parallelization**), and running multiple routines on GPU while keeping data resident on the GPU. (**whole model parallelization**). For the latter approach, communication is only required for model initialization, inter-process communications and output of model grids.

Section I outlines steps needed to setup and configure the F2C-ACC compiler. Section II defines the current Fortran language support and Section III discusses code generation options necessary to build CUDA code. Section IV lists all of the directives currently supported and provides syntax for each. Finally, this compiler generates most of the CUDA code needed to execute code on the GPU, however hand-coded modifications may be required in some instances. Section V discusses some analysis capabilities added in Version 3, and Section VI highlights some programming examples including the use of modules.

<u>New Capabilities in Version 4</u>

- Added support for Fortran modules with the following restrictions:
  - Module variables used in GPU kernels must be copied explicitly via a copyIn or copyOut subroutine in which variables are passed to the given routine via argument list.
  - Subroutines in modules that become GPU kernels are not callable from Fortran CPU code.
- Added automatic generation of data arguments required for each GPU kernel defined by ACC$REGION BEGIN/END directives.

Table of Contents

I. Setup and Configuration

1. Untar the gzipped file, and type "make" to build the compiler.
2. Install the m4 library (http://www.gnu.org/software/m4/).

   NOTE: Generated code relies on the m4 macro library to handle different operating system calling conventions between Fortran and C, collapse multi-dimensional arrays, and support some Fortran intrinsic functions.

3. A makefile is provided in the examples directory for building CUDA or C source code.
   a. Edit the Makefile as needed to define the location of your installed software. (HOME, CUDA, etc).
   b. Suffix runs are provided to generate .o (CPU objects), .cu (GPU objects), and .m4 targets.

4. Download the CUDA toolkit from the NVIDIA site and install:
   http://developer.nvidia.com/object/gpucomputing.html - -> CUDA Downloads

5. Read the CUDA Programmers Guide for information about GPU programming
   http://developer.nvidia.com/cuda-toolkit-32-downloads

6. Build run-time tests in the examples directory
   a. cd examples/gptl_lite; make; cd ..
   b. make [driver_ftn] [driver_cpu] [driver_gpu]
      i. driver_ftn - Fortran only executable
      ii. driver_cpu - Fortran driver + F2C-ACC generated CPU routines
      iii. driver_gpu - Fortran driver + F2C-ACC generated GPU routines

```
> ./driver_gpu
Data Movement Tests
   ACC$DATA Test:         PASSED

 Correctness and Accuracy Tests
   Intrinsics Test:       PASSED
   Scalar Test:           PASSED
   Module Use Test:       PASSED

 Memory and Performance Tests
   Global2D:               PASSED
   Global2DShared1D:       PASSED
   Global2Dloops:          PASSED
   Global2DShared1Dloops:  PASSED
   SharedChunk:            PASSED
   Global3D:               PASSED
   Global3DShared1D:       PASSED
   Promote2DBlock:         PASSED
   Promote2DThread:        PASSED
   Shared3Dout:            PASSED

              ROUTINE   MIN TIME(sec)  MAX TIME(sec)
             global2D        0.010          0.010
     global2Dshared1D        0.003          0.003
        global2Dloops        0.010          0.010
 Global2DShared1Dloop        0.003          0.003
          sharedChunk        0.015          0.015
             global3D        0.009          0.009
     global3Dshared1D        0.003          0.003
       Promote2Dblock        0.016          0.016
      Promote2Dthread        0.015          0.015
          Shared3Dout        0.001          0.001
```

II. Language Support

F2C-ACC supports the most commonly used language constructs for Fortran 77 and Fortran 90, including most declarations, do-enddo, do-continue, if-else-endif, data, parameter, assignment statements, and memory allocation. Language constructs not supported are the character, complex and derived types, all I/O statements, modules, while, where, forall, and select statements, and many Fortran intrinsic functions. In the event a language construct (eg. "interface" statement) is not supported, F2C-ACC will generate the following message that identifies the line in question:

```
F2C-ACC ERROR:  4,25 "Language construct not supported."
```

In this case, the error was on line 4, column 25. The message will be embedded directly in the output text so users can modify the generated C or CUDA code as necessary.

The following intrinsic functions are currently supported:
`abs, acos, asin, atan, ccos, cos, csin, exp, log, max, min, pow, sign, sin, and tan`.

Other functions can be added on request. In the event a mapping between a Fortran intrinsic and C function is not available, or is not supported, a message will be generated that identifies the line in question:

```
ERROR” 5,25 “Fortran intrinsic not supported.”
```

Where the line number and column number are given as 5 and 25 respectively in this example.

User defined kinds are not supported in this release but will be added in version 4.3.

III. Building Code

Two general types routines are supported: kernel routines that run only on the GPU, and device routines that are executed on the GPU but launched from the CPU. Code generation is available thru a script called F2C-ACC. The following run-time options are supported:

--comment	Retain original array statements as comments
--Analysis=[1][2]	Generate analysis information where, 1 = DATA MOVEMENT, 2 = VARIABLE USE
--Constant [Var1=Value] [Var2=Value]	Constant name / value pairs used for local variables
--Fixed	Input files are f77 or f90 fixed format
--Free	Input files are Fortran 90 free format
--Generate=[C][CUDA]	Language Options: CUDA and C, default is CUDA
--OutputFile=<filename>	Name of the generated output file

Generated code relies on the GNU m4 library (http://www.gnu.org/software_m4/) to resolve array references, handle some intrinsic functions, and to resolve platform specific subroutine and function calling conventions.

The examples directory of this distribution contains a sample Fortran subroutine, a Makefile, and a README containing instructions for building source code.

IV. Parallelization Directives

The following directive are supported:

• ACC$DATA	moves data between the CPU and GPU
	allocates GPU constant and shared memory variables
• ACC$DO	identifies block and grid level parallelism for Fortran do-loops
• ACC$INSERT	inserts Fortran into the input source code before translation is done
• ACC$INSERTC	inserts C or CUDA code into the generated code
• ACC$REGION	defines regions of code to be executed on the GPU
• ACC$REMOVE	identifies code that will be ignored by F2C-ACC
• ACC$ROUTINE	indicates how the routine will be called and used
• ACC$SYNC	used for thread synchronization
• ACC$THREAD	restricts execution to a single thread

Directive Continuation Lines

Free Format:

- Place an & at the end of the directive segment
- Use !ACC$> to continue the line, and continue using & as required for successive lines

```
      !ACC$REGION(<nx>,<nz>,        &
      !ACC$>      <var1:in>         &
      !ACC$>      <var2:in>) BEGIN
```

Fixed Format:

- Use CACC$> or !ACC$> for each continuation line

```
      CACC$REGION(<nx>,<nz>,
      CACC$> <var1:in>
      CACC$> <var2:in>) BEGIN
```

ACC$DATA

This directive defines and allocates GPU memory, and can be used to copy data between the CPU and GPU. The **intent** argument, data can be defined, allocated, and copied between the CPU and GPU. Global, constant, local and shared memory are specified using the type argument to allocate memory on the GPU.

References to allocated GPU global memory stored on the CPU for use by one or more kernel routine as needed. The visibility of the pointer can be expressed using the scope argument.

Syntax:

!ACC$DATA (< vars: intent *[, type]*>, [< vars2 ... >])

Required Arguments

vars		comma separated list of variable names
intent		intent of variables listed, options are:
	in	define, allocate and copy data to the GPU
	out	copy data from the GPU to CPU memory
	none	define and allocate memory on the GPU

Optional Arguments

type		*(default is global)*
	constant	the variable uses GPU constant memory, is visible to all kernels in the file. Only scalar constants are permitted..
	local	the variable uses GPU global memory, and visible to the routine where it appears.
	global	the variable uses GPU global memory, and is visible to all routines (via the extern scope in the ACC$REGION directive)
	shared	the variable uses GPU shared memory which is visible to all routines in the file, but must be declared as shared in the ACC$REGION directive.

Notes:

- Constant memory is persistent for the duration of program and is available to all routines in the file.
- Global memory is persistent for the duration of the program and is available to all routines.
- Shared memory is persistent for the duration of an ACC$REGION and is available to all routines in the file where the variable is used.
- Local memory is persistent for the execution of the kernel and is available to all routines in the given file.

Examples:

1. Defines a pointer and allocates GPU global memory. The two examples are equivalent.

```
!ACC$DATA(<flux: none>)
!ACC$DATA(<flux: none, global>)
```

2. Allocates shared memory on the GPU. Dimensions of the given variables must be statically defined. If the dimension is a variable constant, an upper-case name must be given using the –-Constant option.

```
F2C-ACC –Constant NX=50 decls.f90
```

```
decls.f90

parameter (nx = 50)
real :: flux(nx)
!ACC$DATA(<flux: none, shared>)
```

3. Declare U and V as GPU resident variables using the ACC$DATA directive. The **in** keyword indicates data is copied from the GPU to CPU. The **global** keyword makes the variable visible to other files and routines. Use the ACC$REGION to access the variables u and v that are resident on the GPU where **none** indicates no copies between GPU and CPU are needed. The **extern** keyword indicates the variable is defined in another file (in this case: copyIn.f90).

```
copyIn.f90:
   !ACC$DATA(<u,v:in,global>

sub1.f90:
   subroutine s1( nx,nz u, v )

   real, intent (IN) :: nx,nz
   real, intent(IN) :: u(nx,nz)

   !ACC$REGION(<NZ>,<NX>,<u,v:none,extern>,<nx,nz:in>) BEGIN

      <calculations using u & v>

   !ACC$REGION END

   end subroutine s1
```

ACC$DO

Defines loops level parallelism for the GPU device. Two types are currently supported: VECTOR and PARALLEL. The qualifier VECTOR is used for thread parallelism, and PARALLEL is used for block level parallelism. The directives are placed in the code immediately before the do-loop to be parallelized.

Syntax:

```
!ACC$DO VECTOR (dim [ , istart, istop[ base-t ] ] )
!ACC$DO PARALLEL  (dim [, istart, istop ] )
```

Required Arguments

dim:	identifies the dimension of the thread or block parallelism

Optional Arguments

istart	start index
istop	stop value
base-t	thread 0 index

Examples:

1. Using the keyword PARALLEL, identify the next do-loop as block calculations executed by the first dimension of the grid block.
2. Using the keyword VECTOR, identify the next do-loop as thread calculations to be executed by the first dimension of the thread block.

```
real var3D(nz,nx,ny)

!ACC$REGION(<nz>,<nx,ny>) BEGIN
!acc$do parallel(1,2:nx-1,1)
 do i=2,nx-1
!acc$do parallel(2,2:ny,1)
  do j=2,ny
!acc$do vector(1,3:nz,2)
   do k=3,nz
     var3D(k,i,j) = 0.0
   enddo
  enddo
 enddo
!ACC$REGION END
```

ACC$INSERT

This directive is used to insert additional Fortran statements into the original source code before translation C or CUDA is done. For example, nvcc does not support kernel routines, defined in one file and called in another; they must exist in the same file. To overcome this restriction the ACC$INSERT directive is used as follows:

```
File:  sub1.f90

    ACC$INSERT include “my_function.cu”
    subroutine sub1()

      call my_function() ! function defined in my_function.cu

    end subroutine sub1
```

ACC$INSERTC

In contrast to ACC$INSERT, this directive inserts CUDA or C code directly into the generated code. This directive is useful for getting around limitations of the F2C-ACC compiler, such as including print statements, #include files, initializing the GPU, and other statements.

```
File: sub1.f90

    subroutine sub1()

    !ACC$REGION(<nz>,<nip>) BEGIN
    !ACC$DO VECTOR(1)
    do k=1,nz
    !ACC$DO PARALLEL(1)
      do i=1, nip
        a(k,i) = a(k,i)**2/ik
    !ACC$INSERTC    if ((threadIdx.x ==0) && (blockIdx.x == 0)) {
    !ACC$INSERTC       printf(“a(1,1) = %f\n”,a(1,1));
    !ACC$INSERTC    }
      enddo
    enddo
    !ACC$REGION END
    end subroutine sub1
```

ACC$REGION

Defines a region for GPU acceleration. Generated code will copy all data with intent (IN) or (INOUT) to the GPU, and copy all data with intent (OUT) or (INOUT) back to the CPU after completion of the GPU kernels, unless the intent for each variable is specified.

Syntax:

```
!ACC$REGION ( <threads [thread_option]>,<blocks >
   [ ,<var1 [ => my_var1] : intent, scope, mote >] [, <var2 ... >] )  BEGIN
!ACC$REGION END
```

Required Arguments

threads	number of threads in one or two dimensions
blocks	number of blocks in one, two or three dimensions

Optional Arguments – referred to as the data section

Thread_option

chunk	threading and blocking will applied to a single dimension
blocks=factor	blocking factor or multiple of the given thread dimension

var1	variable name
my_var1	map variable to another variable (see example 4)

intent — intent of variables listed

in	copy data to GPU before the kernel is executed
out	copy data to CPU after kernel has executed
inout	copy to the GPU before and to CPU after the kernel completes
none	data is GPU resident, no copies are needed

scope — defines the type of variable (default is global)

extern	assumes the variable has been defined in another routine or file
local	use GPU local memory, scope restricted to a single executing thread
global	use GPU global memory, data is allocated from the CPU
shared	use GPU shared memory, where the intent applies to copies between GPU global and shared memory

mote
- promote (dim)
  variable promotion (see examples/PromoteDemoteTests.f90)
- demote (dim [, dim])
  variable demotion (see examples/GlobalSharedTests.f90)

Notes:

- The number of threads must be equivalent to the variable dimension over which the loop calculations are done.
  - The optional thread argument ‘chunk’ allows blocking and threading over a single array dimension. See Section VI for details.
  - Optional block=factor permits multiple blocks to be combined in a thread block. Se section VI for details.

- The following C programming language defaults apply:
  - all scalars are pass-by-value, and arrays are pass-by-reference, unless otherwise specified.

- The optional arguments allow the programmer to list variables used in the region, and give their intent and scope. If these optional values are not given, they are determined automatically thru variable analysis. In cases where intent is not explicitly given (via the intent attribute), variables are assumed to be local to the CPU and require intent inout. No inter-procedural analysis is done.

Examples:

1. `Data Movement`**:** `U3d` is copied to the GPU using ACC$DATA directive. The ACC$REGION directive designates the **local** variable `flux` to GPU prior to executing the kernel; u3d with intent **none** is a GPU resident variable and no data copies between CPU and GPU are required.

```
copyIn.f90:
   !ACC$DATA(<u3d:in>)

flux.f90:
   !ACC$REGION(<nz>,<nip>,<flux:in,local>,<u3d:none,extern>)
   !ACC$> BEGIN
      < stmts using flux and u3d>
   !ACC$REGION END
```

2. Variable remapping may be needed to support program execution where `s1` is called two times to perform calculations on two variables: `u` and `v`. A variable may need to be added to the subroutine to discern between the invocations of `s1`. In this example, ACC$DATA is used to declare `u` and `v` on the GPU and ACC$REGION users remaps the variable `var` to `u` or `v` depending on the value of `callSite`.

```
copyIn.f90:
   !ACC$DATA(<u,v: none, global>
   call s1 (u,1)
   call s1 (v,2)

sub1.f90:
   subroutine s1( var, callSite)

   real var(nz,nx,ny)
   !ACC$REGION(<var=>u[callSite==1],var=>v[callSite==2]) BEGIN
      <calculations with var>
   !ACC$REGION END
   end subroutine s1
```

3. GPU Shared Memory: The ACC$DATA directive must be used to define shared memory variables. Shared memory is allocated statically, so the array dimensions must be a constant. If no sizes are specified, the size of original array will be used. Use F2C-ACC --Constant var=dim to define dimensions if required.

```
sub1.f90
   real :: u3d(nz,nx)
   !ACC$DATA(<u3d:none,shared>)

F2C-ACC --Constant nx=1000,nz=50 sub1.f90
      - defines NX=1000, NZ=50 and __shared__ float u3d(NZ,NX)
```

4. GPU Shared Memory Data Movement: The intent of a global or local variable manages data copies between GPU and CPU. In contrast, the intent of a shared memory refers to data copies between GPU shared and global memory. Entries for both CPU-GPU and global-shared may exist in an ACC$REGION directive.

   Note: Only intent none is currently supported; copies between GPU global and shared memory will be added in a future version. The example designates a CPU to GPU copy, given by intent **in**, of u3d. An additional copy from GPU global memory to shared memory is specified by intent **in** for the shared entry: <u3d:in,shared>.

```
copyIn.f90:
   !ACC$DATA(<u3d:none,shared(50,1000)>)

flux.f90:
   !ACC$REGION(<nz>,<nx>,u3d:in,local>,<u3d:in,shared>) BEGIN
      < stmts using u3d>
   !ACC$REGION END
```

ACC$REMOVE

This directive is used to remove Fortran text before it is translated. ACC$REMOVE is often used in conjunction with ACC$INSERT to replace Fortran with valid CUDA or C. The directive also helps get around limitations of the F2C-ACC compiler, while preserving the original Fortran code.

Syntax:

```
!ACC$REMOVE BEGIN
!ACC$REMOVE END
```

ACC$ROUTINE

This directive identifies the way in which the function or subroutine will be executed: as C routine called by Fortran, as a C routine that executes on the CPU (and could launch GPU routines), or as a routine that executes on the GPU when CUDA is specified, or on the CPU when it is not.

This directive must be placed just before the subroutine or function.

Syntax:

```
!ACC$ROUTINE ( type  [ : chunk [= dim] ])
```

Required Arguments

type specifies how the routine is called

FORTRAN	- called from a Fortran routine
CPU	- called from a F2C-ACC C generated routine from the CPU
GPU	- called from a F2C-ACC C generated routine from the GPU; reverts to cpu if C code generation is specified (rather than CUDA)

Optional Arguments

dim	thread / block dimension in which chunking should be done. **NOTE: This option is only valid for the type: GPU**

Examples:

```
!ACC$ROUTINE (GPU : chunk )     ! chunks over the first thread / grid block
subroutine run_on_GPU_only( args .... )

!ACC$ROUTINE (CPU)              !routine is called from C code

!ACC$ROUTINE (FORTRAN)          !routine is called from Fortran
```

ACC$SYNC

This directive is placed in the code to insure thread synchronization within each thread block. Synchronization is needed when (1) there are more threads than can be contained in a single warp (32 threads), and (2) there is a dependency between one or more elements of a variable.

In the following example, the elements of array a will be scheduled to run on a single streaming multiprocessor (SP) of the GPU on three warps (32 threads each). A synchronization is placed between the loops, because `b(33)`, executing on the second warp (threads 32-63), requires `b(32)` which is updated on the first warp.

```
sub sub1()

integer, parameter:: nz=96
real a(nz,nip)

!ACCREGION(<nz>,<nip>) BEGIN

!ACC$DO PARALLEL(1)
do i=1,nip
  !ACC$DO VECTOR(1)
  do k=2,nz
    a(k,i) = c(k,i)**2.
  enddo
enddo

!ACC$SYNC

!ACC$DO PARALLEL(1)
do i=1, nip
  !ACC$DO VECTOR(1,2:nip,1)
  do k=2, nip
    b(k,i) = a(k-1,i)
  enddo
enddo

end subroutine sub1
```

ACC$THREAD

This directive defines GPU calculations to be executed by a specific thread. This directive is generally used for array references to a specific array index, but can also be used to serialize GPU calculations. There are two forms of this directive. The single line directive applies to the next executable statement. The multi-line directive, bounded by BEGIN / END, applies to all statements contained the directive pair.

Syntax:

!ACC$THREAD (thread)

!ACC$THREAD (thread) BEGIN
!ACC$THREAD END

Required Arguments

thread	thread number

Optional Arguments

dim	thread block dimension

Restrictions:

- Must appear within a defined accelerator region (ACC$REGION)
- Threaded region must be terminated before another is started.
- Dim is not currently supported, so the directive only applies only to the first dimension of the thread block.

Examples:

1. Single line statement

```
real a(nz,nx,ny)

!ACC$THREAD (1)
      a(1,i,j) = 0
```

2. Multi-statement executed by thread number 1.

```
!ACC$THREAD (1) BEGIN
do i=1, nx
do j=1, ny
      a(1,i,j) = 0
enddo
enddo
!ACC$THREAD END
```

V. Analysis Capabilities

An upgrade in F2C-ACC version 4.2 automatically generates variable analysis for all variables contained in each accelerator region (!ACC$REGION BEGIN / END). The arguments are based on the assumption that variables passed via subroutine arguments reside on the CPU. When variables are used in an accelerator region, a copy between CPU and GPU is generated (intent IN). When variables are updated in an accelerator region, they are copied from the GPU to CPU at the end of the region (intent OUT). If multiple accelerator regions appear in a single routine, inter-region analysis is done to reduce copies between the CPU and GPU.. The –Analysis=1 option for F2C-ACC will list the intent required for each accelerator region.

To improve variable analysis, it is recommended that the intent attribute be specified for each variable passed via the subroutine argument list.

*A. **Data Movement** (--Analysis=1)*

This option analyzes all variables to determine if they need to be passed between CPU and GPU. Output from the analysis will be a recommended string which can be placed in the data section of the ACC$REGION. For example:

```
ysu.f90 ACC$REGION line 0266 recommended data section arguments are:
    <ims,ime,jms,jme,kms,kme:in>,<a,b,c:inout>
```

If there are no changes required, the following message will be given:

```
ysu.f90 ACC$REGION line 845: Data section is properly defined.
```

Current Limitations:

- Loop variables "updated" in the GPU region are communicated to the CPU
- Constants are included in the string and must be removed
- Intent may not be correct for variables updated within an if-conditional or subroutine

Work is continuing to fix these issues and improve the analysis capabilities.

B. Variable Use *(--Analysis=2)*

This option is designed to look at how frequently each variable is accessed in the GPU regions contained within a single file. The information can be useful for determining likely candidates for using GPU shared memory. Since the scope of **shared memory variables** are file based, the analysis gives the number of accesses for each variable, used in one or more GPU regions. Output is given for variables accessed at least 4 times. Some generated output from this analysis is given:

```
>F2C-ACC --Analysis=2 ysu_f2cacc.F90

Running F2C-ACC Code Analysis ...


F2C-ACC Analysis:  93 arrays used in 6 GPU regions were analyzed.
F2C-ACC Analysis:  52 arrays contained at least 4 accesses:
   1D  variable:       kpbl was used:  28 times -->  1 15  5  4  2  1        Decl: kpbl[its:ite]
   1D  variable:       hpbl was used:  26 times -->  1 17  6  0  2  0        Decl: hpbl[ims:ime]
   2D  variable:       xkzh was used:  26 times -->  0  0 24  1  1  0        Decl: xkzh[its:ite,kts:kte]
   2D  variable:       xkzm was used:  25 times -->  0  0 14  0 11  0        Decl: xkzm[its:ite,kts:kte]
   2D  variable:         zq was used:  21 times -->  8  8  5  0  0  0        Decl: zq[its:ite,kts:kte+1]
   2D  variable:         ux was used:  20 times -->  2  9  4  0  4  1        Decl: ux[ims:ime,kms:kme]
   2D  variable:         vx was used:  20 times -->  2  9  4  0  4  1        Decl: vx[ims:ime,kms:kme]
   2D  variable:         qx was used:  18 times -->  2  2  6  6  2  0        Decl: qx[its:ite,kts:kte*ndiff]
   1D  variable:       brup was used:  18 times -->  0 18  0  0  0  0        Decl: brup[its:ite]
   2D  variable:         f1 was used:  18 times -->  0  0  8  1  8  1        Decl: f1[its:ite,kts:kte]
   2D  variable:         za was used:  17 times -->  5 12  0  0  0  0        Decl: za[its:ite,kts:kte]
   1D  variable:     pblflg was used:  17 times -->  1 10  2  2  2  0        Decl: pblflg[its:ite]
   1D  variable:       brcr was used:  16 times -->  0 16  0  0  0  0        Decl: brcr[its:ite]
   1D  variable:        ust was used:  16 times -->  0 13  2  0  1  0        Decl: ust[ims:ime]
   2D  variable:       xkzq was used:  16 times -->  0  0  5 11  0  0        Decl: xkzq[its:ite,kts:kte]
   2D  variable:       thvx was used:  15 times -->  2  9  4  0  0  0        Decl: thvx[its:ite,kts:kte]
```

C. Diagnostic Messages

1. Communications between the CPU and GPU

In the event there are multiple GPU and CPU regions within a single Fortran subroutine, analysis is done to determine if inter GPU / CPU communications is needed. Warning messages are automatically generated if communications is required:

WARNING: <file><line,col> CPU-to-GPU communication needed for the referenced variable.
WARNING: <file><line,col> GPU-to-CPU communication needed for the referenced variable.

To resolve the warning, insert the referenced variable, with intent into the data section of the relevant ACC$REGION directive.

2. Variable storage analysis

Each variable contained in an accelerator region is analyzed to determine if there is sufficient storage for block level (ACC$DO PARALLEL) calculations. There must be an array dimension for block calculations or each block will write to the same storage location, resulting in an incorrect result. The following error will be generated in the event this condition is detected:

ERROR: <file><line,col> Variable requires storage for DO PARALLEL calculations

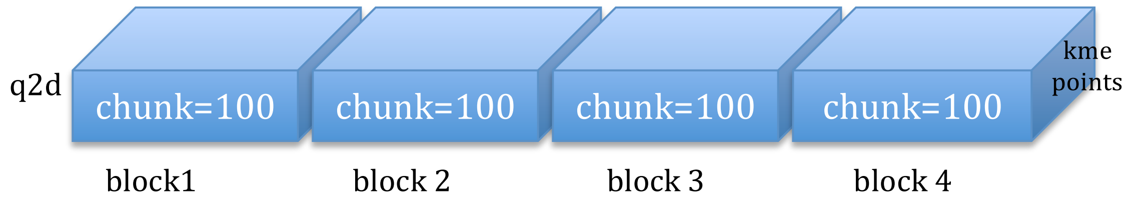
VI. GPU Parallelization Examples

This section highlights more complex parallelization capabilities of F2C-ACC.

Threading and blocking over a single dimension

Parallelism can only be exploited when calculations are not dependent on another element in the same dimension. In general, loops are parallelized for thread-level or block-level parallelization. Normally, one dimension would be assigned the thread-level calculations (ACC$DO VECTOR), and one would be given the block-level calculations (ACC$DO PARALLEL). Sometimes two independent loops do not exist, and thread and block level calculations must be applied to the same array dimension.

In the example below, the variable q2d contains only one independent dimension where parallelism can be exploited. The option **chunk** is used in ACC$REGION to specify that 100 threads will be used for the thread level calculations, and the next 100 points will be calculated by threads in block 2, as illustrated. Note in the example, that the thread size of 100 is used to calculate the number of blocks required for this loop. In the event the total number of points is not evenly divisible by 100 in this example, an additional block is specified for the remaining calculations (`<ime-ims+1/100+1>`). In addition, loop ranges (`ims:ime`) must be specified in ACC$DO VECTOR. Finally, either ACC$DO VECTOR or ACC$DO PARALLEL could have been used in this example and would have generated exactly the same CUDA code.



```
subroutine physics(qv2d,q2d,ims,ime,kms,kme)

implicit none
integer i,k
real   ,intent (INOUT) :: q2d(ims:ime,kms:kme)
real   ,intent (IN) :: qv2d(ims:ime,kms:kme)

!ACC$REGION(<100:chunk>,<ime-ims+1/100+1>,<t3d:in>) BEGIN
do k=kms,kme-1

!either directive can be used, however you must specify the optional
!range values (ims:ime) to restrict execution points in the domain

!ACC$DO VECTOR(1,ims:ime)
  do i=ims,ime
    q2d(i,k) = q2d(i,k+1) * qv2d(i,k)
  enddo
enddo
!ACC$REGION END

return
end subroutine physics
```

Threading over multiple dimensions

If the number of points in a single array dimension is small, threading over more than one dimension can increase the thread count and potentially improve performance. This optimization, specified using the **`chunk`** option in ACC$REGION, can only be done if the block and thread calculations are independent.

The chunk takes an option field that identifies the block dimension in which the chunking will be applied. If no value is specified, then the index of the block dimension will be used. In this example, however, the option **`chunk=1`** means chunking in thread dimension 2, will be applied to the first block dimension. If the number of blocks, `jme,` were not evenly divisible by 2, an additional block would be needed to calculate the remaining j points (eg. <(jme-jms+1)/2+1>) . In addition, range values `jms:jme` also need to be specified to restrict the data elements that are accessed as follows: `!ACC$DO PARALLEL(1,jms:jme)`.

```
subroutine physics(rd,t3d,ims,ime,jms,jme,kms,kme)

implicit none
integer i,j,k
integer, intent (IN) :: nthreads
real   ,intent (INOUT) :: t3d(ims:ime,kms:kme,jms:jme)
real   ,intent (IN) :: rd       ! gas constant
integer hpts

!ACC$REGION(<(ime-ims+1),2:chunk=1>,<(jme-jms+1)/2>,<t3d:in>) BEGIN
!ACC$DO PARALLEL(1)
do j=jms,jme
  do k=kms,kme
!ACC$DO VECTOR(1)
    do i=ims,ime
      t3d(i,k,j) = t3d(i,k,j) * rd
    enddo
  enddo
enddo
!ACC$REGION END

return
end subroutine flux
```

Using modules

Declarations that appear in a module are built with the Fortran source code. If you wish to use variables defined in a module via a use statement, you must define a routine to copy data to the GPU. Variables must be copied via argument list to the copy routine and the ACC$DATA directive must be used to define pointers to CPU and GPU memory, allocate storage, and copy data as required to the GPU. A working example of a copy routine can be found in examples/copyIn.f90.

Once data is resident on the GPU, declarations are listed in ACC$REGION with the **extern** qualifier. This indicates that a pointer to GPU memory should be used. Similarly, for variables used in the CPU generated C code, an additional pointer to CPU memory is available. A working example is shown in: examples/use_module.f90 that references the module m1 which contains variables also in the copy routine.

VII. Limitations and Known Bugs

F2C-ACC is not able to generate all of the code translations necessary to run code on the GPU. There are several known areas where code modifications are required.

1. Fortran 90 Array Assignments

Array assignments must contain loops so the ACC$DO directive can be used to express thread (VECTOR) or block (PARALLEL) level parallelism.

Original Source

```
real a(nz,nx,ny)



a = 0.
```

Modified Source

```
real a(nz,nx,ny)
do k = 1, nz
 do i = 1, nx
  do j = 1, ny
    a(k,i,j) = 0.
  enddo
 enddo
enddo
```

2. Modules are not currently supported. Replace use statements in the code by explicitly passing arguments via each subroutine or function argument list.

3. I/O statements are not permitted. Use !ACC$INSERTC and !ACCREMOVE to replace Fortran I/O with C I/O.

Original Source

```
read(6) a,b,c,d
```

Modified Source

```
!ACC$REMOVE BEGIN
read (6) a,b,c,d
!ACC$REMOVE END
!ACC$INSERTC  fread(6) a,b,c,d
```